YADAS: An Object-Oriented Framework for Data Analysis Using Markov Chain Monte Carlo LA-UR-01-4804

Todd L. Graves

August 17, 2001

1 Introduction

In this paper I discuss a system written in Java for performing Bayesian data analyses. The goals of this system are to minimize the amount of programming time necessary to start Markov Chain Monte Carlo (MCMC) algorithms running. If necessary we are willing to incur increased processing time if we save our own time. An object-oriented framework has many advantages for constructing a general-purpose Bayesian analysis system: roughly speaking, by modifying a single object, one may change a model by adding another level to a hierarchical model, change a distributional form or link function, or change the form of a Metropolis step.

The structure of this paper is as follows. Section 2 is the most extensive section, containing a description of the architecture of the software, serving as a users' manual, and indicating how analysts may construct their own applications by extending the key classes in the existing software.

The acronym YADAS stands for "yet another data analysis system." The author is indebted to Steve Upton for this suggestion.

1.1 Terse review of object-oriented terminology

Object-oriented programming consists of manipulations of *objects*. A *class* is an abstract description of what all objects belonging to that class have in common. An individual object created at run time which is an example of a class is known as an *instance* of that class. Class descriptions include data fields, so that a class is in part a data structure, but they also include methods. A *method* is a function: the thing that makes methods different from non-object-oriented functions is that each method belongs to an object. In addition to the explicit argument list of a method, it implicitly has the object to which it belongs as an argument, and this will often be the most important argument in good object-oriented designs. A special type of method is the *constructor* method of a class, which is called to return a new instance of the class.

Interfaces are collections of methods. A class is said to *implement* an interface if it contains method definitions for all methods contained in the interface. An interface is an important component of an object-oriented design because many otherwise different classes which implement an interface can be manipulated together: for example, one may construct an array of objects that all implement the interface, then loop through this array, calling the interface's methods for all its elements. Additionally, one can write methods whose arguments are objects that implement the interface if the methods treat the arguments only through the methods of the interface. In this way, different classes can be handled using the same code, where if the different classes require somewhat different behavior, this can be taken care of using each class's implementation of a method. This sort of behavior is referred to as *polymorphism*.

Another fundamental concept in object-oriented design is that of inheritance. When a class is *inherited from* (or a *subclass* of) another class (the *superclass*), its data fields are a superset of the superclass's, and it has a superset of the methods of the superclass, although these methods can be redefined in the subclass.

A somewhat advanced topic in Java is the *inner class*. An inner class is defined, and only used, inside another class. When inner classes are appropriate, they can make it easier to understand the code, and they also have the advantage of having access to the outer class's fields and methods. (This concept of differential access to fields and methods is important but not particularly relevant to this paper. Programmers are encouraged to think about which fields and methods in a class should only be called by an instance of the class itself, and declare these fields and methods as *private*; other fields and methods have *public* or some other form of access. The advantage is that another programmer using a class that you have written does not need to pay any attention to the private fields and methods. This is an example of the concept of "hiding the implementation.")

1.2 The `DataFrame` class: loading data

To describe the MCMC-related classes in the remainder of this paper, it is necessary to have a notation to refer to data that have been imported from files. YADAS uses a class called `DataFrame`. It will assign arrays of data of various types to appropriate names. For example, the file `test.dat` contains a pipe-separated rectangular array of data with three header lines as follows:

```
60
y|x|group|n
r|r|i|r
2.43|0.80|0|5
2.19|0.65|0|2
2.99|1.45|1|4
.................
```

The first line indicates that there are 60 lines of data. The next line contains variable names, and the third line contains variable type information ('r' for real data, 'i' for integer, and 's' for strings). The first two columns provide initial values for the `y` and `x` variables; one or both of these variables could be constant in which case the values do not change from their initial values. The third column lists which of two or more groups the data points belong to (in other words, the column helps define a categorical variable). The fourth column, for example, lists how many repeated measurements went into computing the mean value listed in the `y` column; this would later be used in describing the amount of variability in the data points. We now load the data using the command

```
DataFrame d = new DataFrame(``test.dat'');
```

The `DataFrame` class contains many useful methods.

- `r(String name)` returns an array of the variable in the data frame that is named `name`. `i(String name)` and `s(String name)` are analogous but used for integer or string variables respectively.
- `r(double x)` returns an array of the length of the variables in the data frame, all of whose entries are equal to `x`. `i(double x)` and `s(double x)` are analogous.

- `u()` returns an array from zero to one less than the length of the variables in the data frame.

1.3 Some related work

An existing software package available in two incarnations, BUGS and WinBUGS, (Spiegelhalter et al, 1994; Lunn et al, 2000), is very useful for Bayesian data analyses, but it handles only a limited class of models and is not straightforward to extend. Also, it is not possible to call it as a subroutine of other code, as would be necessary in the design problem that uses, for example, a genetic algorithm.

Another package developed at Los Alamos, the Bayes Inference Engine (Hanson and Cunningham, 1999) performs Bayesian analyses of imaging problems. It is an excellent package for radiography problems, but it does not include methods for analyzing more general problems.

2 An Object-Oriented Description of Bayesian Data Analysis

The philosophy embodied in the software is that there are three types of components to a Bayesian data analysis: parameters, probabilistic links between these parameters (which we will refer to as "bonds"), and methods of updating the parameters. In an object-oriented architecture, each of these types of components becomes a class or an interface. In this section we discuss the `MCMCParameter` class, the `MCMCBond` interface, and the `MCMCUpdate` interface, how users can use them in their applications, and how users can extend them to add power to the framework.

2.1 `MCMCParameters`

The first thing an analyst does when performing an analysis using YADAS is construct an array of `MCMCParameters`. First, we discuss what an `MCMCParameter` is from a statistics point of view. The answer will depend on the application, but the rough idea is that all quantities that are updated in the same way and which relate to the other parameters in the same way, will be grouped together in a single `MCMCParameter`. For example, in a regression problem with a univariate response and J predictor variables plus an intercept, the response variable $Y = \{Y_i : 1 \leq i \leq n\}$ is an `MCMCParameter`, the jth covariate $X^j = \{X_i^j : 1 \leq i \leq n\}$ is another `MCMCParameter`, the vector of regression coefficients $\beta = \{\beta_j : 0 \leq j \leq J\}$ is another, and further `MCMCParameters` are the error variance σ^2, and the constants in the prior distributions for β and σ^2. Note that data, constants, and unknown parameters can all be `MCMCParameters`; the only difference is that some of these parameters are updated in the course of the MCMC, while others remain constant. (It is not necessary that constants be included in the model as `MCMCParameters`.)

An `MCMCParameter` can be defined more clearly from the point of view of the software. First, an `MCMCParameter` contains a `value` record (accessible using the method `getValue()`) which contains the (current) value of the parameter. An `MCMCParameter` also includes an array of real-valued step sizes which indicate how large the Metropolis steps the parameter takes should be. This step size array is of the same length as the `value` record, although in many applications all of its entries will be the same.

An example of a definition of an array of `MCMCParameters` is as follows. The expressions involving `d` refer to a `DataFrame` (see §1.2).

```
  MCMCParameter paramarray[] = new MCMCParameter[]
    { x = new MCMCParameter (d.r(``x''), d.r(0.0), ``x''),
      mu = new MCMCParameter (new double[] {1.0, 0.0}, d.r(0.1, 2), ``mu''),
      sigma = new MCMCParameter (new double[] {1.0}, d.r(0.2, 1), ``sigma''),
    };
```

The first parameter is `x`, whose initial values are given by the elements real-valued variable `x` in the `DataFrame` `d` (this is the meaning of `d.r(``x'')`). `x` is data rather than a modifiable parameter, since the second argument to the `MCMCParameter` constructor is an array of all zeroes (for a `DataFrame d`, the `r` method called with a single real-valued argument returns an array of the same length as the variables in `d` with all entries equal to that argument). The second variable, `mu`, has only two values which are initially set to one and zero; presumably this is a two-sample analysis and elsewhere we will indicate which of the data points come from the population with the first mean and which from the second. The two elements of `mu` will take Metropolis steps of the same size, namely 0.1 (`d.r(x, n)` returns an array of length $n$ whose elements are all equal to $x$). The third parameter, `sigma`, has only one value and hence only one step size. In this analysis one assumes homoscedastic errors.

`MCMCParameter` is a class that implements the `MCMCUpdate` interface, which means that it knows how to try to update itself. We will discuss the `MCMCUpdate` interface later, but all `MCMCParameter`s are capable of updating themselves in the following way. `MCMCParameter`s contain an array of step sizes to be used in Metropolis steps. The default procedure for updating an `MCMCParameter` involves looping over its components and generating a Metropolis candidate by adding a random Gaussian amount to a component, with standard deviation given by the Metropolis step size.

The `MCMCParameter` constructor, then, takes three arguments: an array of `double`s to place in the value field, an array of (`double`) Metropolis step sizes, and a `String` which contains the name of the parameter (values of the parameter will be saved to a `.out` file of that name).

The default structure of an `MCMCParameter` doesn't necessarily work for all parameters in all applications. Sometimes it is necessary to subclass `MCMCParameter` to modify the way it behaves (e.g. to change the way it is updated). For example, problems with ordering restrictions on some of the parameters, the `OrderedMCMCParameter` class is useful. `OrderedMCMCParameter`s contain information about how some of the elements are constrained to be less than other elements. This has implications for how the parameter can be updated legitimately and efficiently: in fact, we have proposed four different types of updates that are relevant to `OrderedMCMCParameter`s. Therefore, the author of the `OrderedMCMCParameter` class extends `MCMCParameter`, defines some new fields to hold the ordering information, defines inner classes which implement `MCMCUpdate` to describe these four new update types, and defines the `MCMCUpdate` methods of `OrderedMCMCParameter` to loop through these update inner classes but otherwise to do nothing. Covariance matrices are also a special type of parameter: one needs to know how to take their trace and determinant, and also how to update them in MCMC, for example by proposing moves that preserve positive definiteness. A CovarianceMatrix `MCMCParameter` is under development.

One advantage of the `MCMCParameter` class is that it is extremely easy to modify application code in order to convert a parameter from constant to random: this is done by changing the parameter's Metropolis step size from zero to positive or vice versa. Also, it is very easy to convert an application in which a variance (for example) is constant across data points, to one in which variance is a function of the data point with the help of appropriate `ArgumentMakers` (§2.2.1).

2.2 The `MCMCBond` Interface

After an analyst makes an array of `MCMCParameter`s, the next step is to construct an array of objects implementing the `MCMCBond` interface in order to describe the probabilistic relationships between the parameters. An `MCMCBond` is a relationship between two or more `MCMCParameter`s, typically consisting of a probability density function. By far the most common class implementing `MCMCBond` is `BasicMCMCBond`, which handles most continuous distributions and which will be discussed in detail later.

The `MCMCBond` interface consists of five methods: the `compute()` method with four different argument lists, and the `getParamList()` method. This last returns an `ArrayList` containing all the parameters involved in the bond. The `compute()` methods are used to calculate the bond's contribution to the change in posterior when one or more of the parameters is changed. Each of the four different argument lists handles a way in which the parameters can be changed. The first takes an `int` which indicates which of the parameters will be changed, a `double` which is the new value of one of the elements of the parameter, and an `int` indicating which element is a candidate to be changed. The second takes the `int` argument and then an entire array to be the new value of the parameter. The third is like the second except that it allows multiple parameters to be changed, so that it takes an array of `int`s and a two-dimensional array of `double`s to be the new values. The fourth is like the third, but the number of parameters in the argument is less than the number of parameters in the bond, so that a third argument is necessary to demonstrate which of the bond's parameters will be changed.

2.2.1 `BasicMCMCBond`

One of the most critical classes in YADAS is the `BasicMCMCBond` class, which handles a very large number of types of bond. The intent of this class is to handle a large number of commonly encountered relationships between parameters, and to make it as easy as possible to perform tasks like make a model hierarchical or add a level to the hierarchy, change the distributional form of the relationship, or to change the link function. The constructor of `BasicMCMCBond` takes several arguments:

1. an array of `MCMCParameter`s that are related by the bond;
2. an array of objects implementing the `ArgumentMaker` interface;
3. an object implementing the `Likelihood` interface (e.g. Gaussian, Gamma).

`Likelihoods`. A class implementing the `Likelihood` interface is just a function: it has a `compute` method which takes a two-dimensional array of `double`s and returns the (`double`) value of the likelihood function evaluated at these arguments. Examples of `Likelihood`s are `Gaussian`, `Gamma`, `InverseGamma` (although the inverse gamma distribution can also be implemented using the `Gamma` class and an appropriate `ArgumentMaker`), and `Dirichlet`.

`ArgumentMakers` The `ArgumentMaker` interface is responsible for much of the power and versatility of the `BasicMCMCBond` construct. `ArgumentMaker`s have a single method, `getArgument`, which takes two-dimensional arrays of `double`s, and converts them into one-dimensional arrays of `double`s. The input arrays are typically (always, in a `BasicMCMCBond`) all the `value` records of all the `MCMCParameter`s in the bond, before and after being perturbed by a candidate Metropolis step. The output will go directly into the `Likelihood` function. A compelling example of the `ArgumentMaker` is `LinearModelArgument`, which returns sums of products of pairs of variables. It is trivial to add capability for a link function for generalized linear models: `GLMArgument` will be a subclass of `LinearModelArgument` because it will behave in almost exactly the same way, but when the linear argument is computed, it needs to be run through a link function that will be an additional argument to the `GLMArgument` constructor. Note that different families with identity links can be handled using `LinearModelArgument`s and changing the `Likelihood` function.

Another method in the `ArgumentMaker` interface is a method that computes the log Jacobian to be added to the log likelihood. Also one needs to determine whether a parameter plays the role of data in a bond, so that the Jacobian function is only computed for data.

IdentityArgument The most common `ArgumentMaker` is `IdentityArgument`, which picks one of the variables in the input to `getArgument()` and returns it unchanged. In fact, there is an additional constructor to `BasicMCMCBond` which requires no `ArgumentMaker`s as arguments and which instead constructs one `IdentityArgument` for each parameter in the bond. The constructor of `IdentityArgument` takes a single argument, an integer encoding which variable to return. For example, suppose that we are considering a model with measurement error: a variable called `u` has been measured with error and its true value is called `trueu`. The measurement process generates Gaussian errors of known standard deviation `sdu`. `u, trueu`, and `sdu` are all `MCMCParameter`s. This bond is specified by

```
new BasicMCMCBond (new MCMCParameter[] {u, trueu, sdu},
                   new Gaussian());
```

In this example `u, trueu`, and `sdu` all have the same length, so that the ith element of `u` has mean equal to the ith element of `trueu` and standard deviation equal to the ith element of `sdu`. Another way of saying this is that the `ArgumentMaker`s in this bond are identity functions, returning the values of the three parameters intact. The expression above is shorthand for a more explicit description of the identity arguments:

```
new BasicMCMCBond (new MCMCParameter[] {u, trueu, sdu},
                   IdentityArgument.IdentityArgumentArray(3),
   new Gaussian());
```

here the `IdentityArgumentArray` returns three `IdentityArgument`s, the first of which provides the first parameter, the second argument the second parameter, etc. This last expression is in turn shorthand for

```
new BasicMCMCBond (new MCMCParameter[] {u, trueu, sdu},
                   new ArgumentMaker[] { new IdentityArgument(0),
                                         new IdentityArgument(1),
                                         new IdentityArgument(2) },
   new Gaussian());
```

GroupArgument Almost as common is `GroupArgument`, which is used in the cases where a single component of a parameter is used in the distribution for more than one component of another parameter. (For example, if $X_1, \ldots, X_n$ are normally distributed with common mean μ and common standard deviation σ, a `GroupArgument` is required to expand the scalars μ and σ so that each of the n X's has this mean and standard deviation.) The `GroupArgument` constructor takes two arguments, an `int` describing which parameter to return, and an array of `int`s called an *expander*. If the ith element of the expander is j and if the `which` argument is equal to k, then the ith element of the return value of `GroupArgument` is the jth element of the kth column of the input to `getArgument()`. For example, suppose that we have a sample of values `theta` from a Gaussian distribution with common mean `mutheta` and common standard deviation `sdtheta`.

```
new BasicMCMCBond (new MCMCParameter[] { theta, mutheta, sdtheta },
                   new ArgumentMaker[] { new IdentityArgument(0),
                                         new GroupArgument(1, d.i(0)),
                                         new GroupArgument(2, d.i(0)) } ,
                   new Gaussian());
```

The first element in the array of `ArgumentMaker`s returns the 0th parameter, `theta`, without modification. The second and third `ArgumentMaker`s expand `mutheta` and `sdtheta` so that they are the same length as theta. d.i(0) constructs an array of zeroes of the same length as the variables in d. Since all the elements of this array are zero, mutheta is expanded by taking repeated copies of its 0th element.

ConstantArgument ConstantArgument is used to return arguments that are not modified during the course of the MCMC. Because of this class, it is not necessary to define excessive numbers of MCMCParameters. ConstantArgument has three constructors, one that provides a scalar constant, one that provides an array of identical scalar constants, and one that uses an array of doubles read, for instance, from a file. For example, to say that the MCMCParameter alpha of length n has a Gaussian prior distribution with mean given by the variable named mualpha in the DataFrame d1 and with common standard deviation 1,

```
new BasicMCMCBond (new MCMCParameter[] { alpha },
                   new ArgumentMaker[]
                 { new IdentityArgument (0),
                           new ConstantArgument (d1.r(‘‘mualpha’’)),
                           new ConstantArgument (1, n) },
                   new Gaussian ());
```

LinearModelArgument This class makes it possible to do all sorts of linear regressions using only the Gaussian likelihood function, or also to change a linear model to a generalized linear model by changing only one statement. Its interface is in a state of flux, but it currently takes arguments that indicate whether or not the model contains an intercept, which parameters to multiply by each other before summing, and how to expand these parameters. For example, suppose that we are fitting a model to results from auto races on several different tracks, in which driver abilities are allowed to improve linearly over time: the ability of driver i in year j is $\theta_{ij} = \alpha_i + j\beta_i$. There is also a variance parameter, ϕ, which depends on the track. This is expressed in YADAS as follows.

```
bondarray[0] = new BasicMCMCBond (
   new MCMCParameter[] {alpha, beta, year, phi},
   new ArgumentMaker[]
      { new LinearModelArgument(true, new int[] {0, 1, 2},
               new int[][] { d.i("driverid"), d.i("driverid"),
               d.u(d.r("yearid").length) }),
        new GroupArgument(3, d.i("trackid")) },
   new AttritionLikelihood( d.i("raceid"), d.i("finishid") ));
```

Four parameters go into this bond: alpha, beta, year, and phi. year is the covariate (note that it would also be possible to write LinearModelArgument so that year could be handled as a constant instead of as a parameter). The likelihood function is a form intended for rank data of this form; see [Stern, 1990] and [Graves et al, 2001], but in order to appreciate the LinearModelArgument, it is not necessary to understand the likelihood function other than the fact that it requires a θ and a ϕ argument. The true signifies that the linear model includes an intercept, while the new int[] {0, 1, 2} signifies that the zeroth, first, and second parameters (alpha, beta, year) in the bond contribute to the linear model. The next argument, the two-dimensional array of ints, is an array of expanders: alpha and beta contain one coefficient for each driver and need to be expanded to the length of the data set, which contains one observation for each finishing position in each race. d.i(‘‘driverid’’) is an array of ints extracted from the DataFrame d; if the ith element of this array is equal to j, then the jth driver corresponds to the ith data point. The jth element of alpha and beta refer to the jth driver.

FunctionalArgument FunctionalArgument is another versatile class: in fact, one can implement linear model arguments using it. After expanding parameters into equal-length arrays, it obtains the ith component of the output by applying a function to the ith components of all the inputs. The function is specified in a class implementing the `Function` interface, which will typically be specified anonymously. (The `Function` interface has a single method, `f(double[] args)`). An example here is definitely in order. Suppose that there is a linear-like relationship between three variables u, v, and w: $(u/v) = \alpha + \beta(w/v) + \epsilon$, where ϵ is Gaussian noise with standard deviation θ. Further suppose that there are two or more populations, each of which has its own value of (α, β, θ). Then we need to construct several arguments: u/v, $\alpha + \beta(w/v)$, and θ, in order to send them through a Gaussian likelihood.

```
new BasicMCMCBond
   ( new MCMCParameter[] { u, v, w, alpha, beta, theta },
     new ArgumentMaker[]
        { new FunctionalArgument (n, 6, new int[] {},
              new int[][] { new int[] {} },
              new Function () {
                  public double f (double[] args) {
                     return args[0] / args[1];
                  }
              }),
          new FunctionalArgument (n, 6, new int[] { 3, 4 },
              new int[][] { d.i(``pop''), d.i(``pop'') },
              new Function () {
                  public double f (double[] args) {
                     return args[3] + args[4] * args[2] / args[1];
                  }
              }),
          new GroupArgument (5, d.i(``pop'')) },
     new Gaussian () );
```

The first `FunctionalArgument` is in the business of computing u/v. In its definition, n and 6 refer to the length of the u/v array and the number of parameters in the bond. The next two arguments define how the parameters in the bond will be expanded: since they are empty, no parameters need to be expanded in order to compute u/v (there is a unique u and v for each data point). Last comes a definition of an anonymous class implementing the `Function` interface: its `f` method computes the ratio of its first two arguments, namely u/v. The second `FunctionalArgument` computes $\alpha + \beta(w/v)$: this time, the third and fourth parameters, `alpha` and `beta`, need to be expanded, explaining the `new int[] { 3, 4 }`. `alpha` and `beta` use the same expander called "`pop`": if the ith data point belongs to population j, then the ith element of `d.i(``pop'')` is equal to $j$. The second `Function` computes $\alpha + \beta(w/v)$. Finally the third argument is a `GroupArgument` which expands the variance parameter θ.

2.2.2 How `MCMCBond`s are used

`MCMCBond`s exist in order that `MCMCUpdate`s can loop over arrays of them to compute the likelihoods of the current parameters and of the candidate new parameters. Therefore, the concept is very simple but very powerful. While the computations that result are not optimally efficient (the code does not perform symbolic manipulation to simplify differences of equations), the code only computes the bonds that contain the parameter(s) potentially being changed.

When making a slight modification of an analysis, it is often as simple as adding one line of code in which an additional `MCMCBond` is defined. Models can also be perturbed by changing the `Likelihood` object or one of the `ArgumentMaker`s.

2.3 The `MCMCUpdate` Interface

The `MCMCUpdate` interface consists of a single method, `update()`. The purpose of classes that implement this interface is to choose a candidate point in parameter space to move to, and to decide whether to move there. The simplest example involves `MCMCParameter` itself, as this class implements `MCMCUpdate` ("each parameter knows how to update itself"). Often it is useful to choose candidate points by modifying several parameters simultaneously, particularly when these parameters are highly correlated. We provide the `MultipleParameterUpdate` class and a couple of subclasses in order to make it easier to update multiple parameters.

Many classes that implement `MCMCUpdate` divide the functionality in `update()` into several other methods. These methods are `candidate()`, `relevantBonds()`, `acceptanceProbability()`, and `takeStep()`, and are nonessential but can help organize update steps in useful ways.

- `candidate()`. `update()` typically calls this method to generate a candidate set of parameters to which the algorithm may or may not step. Most often, this candidate is generated by adding a random Gaussian amount, with standard deviation given by the parameter's Metropolis step size, to a single parameter.
- `relevantBonds()`. This method returns an array of `MCMCBond`s that need to be computed in order to decide whether or not to accept the step to the candidate. If a single parameter is being changed, this is straightforward since `MCMCParameter`s keep records of which bonds they are involved in.
- `acceptanceProbability()`. After calling `candidate()`, `update()` normally calls this method in order to compute the probability that the candidate will be accepted. This method needs to call `relevantBonds()` to compute this probability correctly.
- `takeStep()`. If a random uniform generated in `update()` is less than `acceptanceProbability()`, this method actually changes the parameter(s) in the appropriate way.

These methods are useful because they organize the updating process and therefore make it easier to add functionality by subclassing classes that contain them. For example, some parameters will work better if their Metropolis steps are chosen multiplicatively. To implement such a Metropolis step, one writes a subclass in which `candidate()` proposes a multiplicative step, and in which `acceptanceProbability()` returns a Jacobian multiplied by the superclass's `acceptanceProbability()` return value.

2.3.1 example: `MCMCParameter`

`MCMCParameter` itself implements the `MCMCUpdate` interface, so that all parameters have default ways of updating themselves. Since `MCMCParameter`s often have multiple components, it is necessary to update each of these different values. To this end `MCMCParameter` keeps a `private` variable keeping track of which component is currently being updated. `update()` loops over the components, calling `candidate()` for each component, which adds a Gaussian amount to the current value of each component value. `acceptanceProbability()` then loops over all the bonds (returned by `relevantBonds()`) containing the parameter, evaluating the difference in log likelihood between the new parameter and the old. If the step is accepted, `takeStep()` changes the appropriate entry in the parameter's `value` record.

If there are no unusual update steps in an analysis, the `main()` method of the YADAS application will typically contain a line such as

```
MCMCUpdate[] updatearray = new MCMCUpdate[] { mu, sigma, theta, phi };
```

listing all the updatable parameters in an array.

2.3.2 example: `MultipleParameterUpdate`

The function of a `MultipleParameterUpdate` is to update two or more parameters (or even multiple components of a single parameter) in a single Metropolis step. Therefore the constructor of the `MultipleParameterUpdate` class takes two arguments: an array of `MCMCParameter`s (some subset, presumably all, of these parameters, will be perturbed to generate the candidate), and an object implementing the `Perturber` interface.

It is not yet clear in what MCMC applications `MultipleParameterUpdate`s are useful. However, YADAS provides a promising framework for research into their usefulness.

`Perturbers` Inside the `candidate()` method of the `MultipleParameterUpdate` class is a call to the `perturb` method of the class implementing the `Perturber` interface. The `Perturber` interface consists of two methods. First is `perturb`, which takes a two-dimensional array of `double`s (it will change some of the components of this array) and an integer whose job it is to indicate which components of the array should now be updated. The way the `int` does this is dependent on the type of `Perturber`: a common example is where all the parameters have the same length, and where the ith update attempts to modify the ith element of each of the parameters. The second method in the interface is `numTurns()`, which indicates how many possible values the integer argument to `perturb` can take on.

`InterceptSlopePerturber` One example of a `Perturber` arises in regression, where intercept and slope parameters can be highly correlated. Suppose the analysis includes several separate simple linear regressions, where the intercepts for the regressions arise from a hierarchical model, as do the slopes. Perhaps the typical values of the covariate are very different in some regressions than in others. In an example from auto racing, we had race results from five consecutive seasons, 1996 through 2000. We wished to allow drivers to improve (or regress) linearly in time. Denote the ability of driver i in a race in year j by θ_{ij}; then we assume that $\theta_{ij} = \alpha_i + (j - 1998)\beta_i$. The problem with this formulation is that some drivers raced only in one year. For example, if driver i raced only in year 2000, the data provide information about α_i and β_i only through the linear combination $\alpha_i + 2\beta_i$, so that α_i and β_i will be highly negatively correlated. More generally, suppose that the average value of the years in which a driver competes is $1998 + k$; then a useful update step changes α and β simultaneously while preserving $\alpha + k\beta$. The class `InterceptSlopePerturber` is designed to handle this case. This perturber is implemented so that the larger of the two perturbations is of the size specified in the step size argument. For example, if $k > 1$, then the candidate perturbation is $\alpha \leftarrow \alpha + \sigma Z, \beta \leftarrow \beta + \sigma Z/k$ (Z denotes a random standard Gaussian). If $k < 1$, the candidate perturbation is $\alpha \leftarrow \alpha + k\sigma Z, \beta \leftarrow \beta + \sigma Z$.

The `InterceptSlopePerturber` constructor takes three arguments. The first argument is an array of two integers: the first shows which component of the two-dimensional array sent to `perturb` is to play the role of the intercept, the the second shows which is to act as the slope. The second argument is an array of `double` step sizes: the array is in principle the same length as the return value of `numTurns()`. The third argument contains the values of the quantity k described in the previous paragraph; these will normally be average values of the covariate. If one of these average values is zero, no update occurs. For example, consider the auto racing problem with driver abilities that change linearly. The complete array of `MCMCUpdate`s is specified as follows.

```
MCMCUpdate[] updatearray = new MCMCUpdate[]
    { alpha, beta, phi,
      new MultipleParameterUpdate
          ( new MCMCParameter[] {alpha, beta},
            new InterceptSlopePerturber
```

```
                ( new int[] {0, 1}, mss.r("abmss"), mss.r ("ratio"))),
      };
```

This statement says that four different update steps: the ordinary individual Metropolis steps for `alpha,` `beta`, and `phi`, as well as a step in which `alpha` and `beta` are updated simultaneously. The three arguments to the `InterceptSlopePerturber` are an `int` array showing which parameters in the update array are to be updated, an array of Metropolis step sizes (here extracted from the `DataFrame` called `mss`), and the array that defines the ratios of the steps taken by the first and second parameters.

`AddCommonPerturber` Another common situation is when two parameters are highly correlated because, explicitly or implicitly, one is very nearly a constant plus another. The class to handle this is `AddCommonPerturber`, which is somewhat complicated and sometimes slow enough to execute so that it is a better idea to run the chain for additional iterations and then skip some of them. The difficulty comes in because these parameters are of different lengths and need to be expanded.

In the following example `alpha` is a sample from a population of parameters with mean `mualpha`. Suppose the remainder of the model is such that all of the `alpha`s tend to remain close to `mualpha` and to each other. As a result, `mualpha` is unable to move freely and therefore ends up with an estimated posterior distribution that is not consistent with its prior. We therefore wish to add a common constant to `mualpha` and each of the `alpha`s in a Metropolis step.

```
new MultipleParameterUpdate (
   new MCMCParameter[] {alpha, mualpha},
   new AddCommonPerturber ( 2, alpha.length(), 1,
                            new int[] {1}, new int[][] { d.i("test") },
                            new double[] { 0.2 } )),
```

The six arguments for the `AddCommonPerturber` constructor are the number of parameters, their (expanded) length, which parameter is the dominant parameter (see below), which parameters need to be expanded, their expanders, and an array of step sizes. The "dominant" parameter is the parameter which governs how many distinct update steps are implemented. Above, the scalar `mualpha` is dominant, which means that this update consists of a single step, in which we add a common constant to all of the `alpha`s as well as `mualpha`. If on the other hand `alpha` were dominant, we would first attempt to add a constant to the first element of `alpha` as well as to `mualpha`, then we would attempt to add something both two the second element of `alpha` as well as to `mualpha`, and so on. (It is perhaps difficult to think of a situation in which any but the shortest parameter is dominant.)

2.3.3 example: `OrderedMCMCParameter`, continued

Simple Metropolis steps are not always sufficient for good MCMC performance. Another more complicated update class is used to update parameters whose components are unknown but whose relative order is known. Parameters of this form can benefit from at least four different types of update steps. For this reason the `OrderedMCMCParameter` class, of which X is an example, defines four inner classes that each implement the `MCMCUpdate` interface and that each try to update the `OrderedMCMCParameter` in different ways. Inner classes are useful in this context because instances of inner classes have access to the outer class's fields.

- `IndividualUpdate` attempts to update individual components of the parameter. It determines the upper and lower limits that the component must stay within in order to satisfy the ordering restrictions. Then it adds a Gaussian amount to the component, rejecting the step immediately if the new value is outside the range. Otherwise the Metropolis step proceeds as usual.

- `ShiftUpdate` adds a common constant to all the X's corresponding to a single race. This update together with `ScaleUpdate` given below was intended to speed the convergence of the center and scale of the X's.
- `ScaleUpdate` rescales the $X's$ corresponding to a single race, by the mapping $X_{ij} \leftarrow \bar{X}_i + b(X_{ij} - \bar{X}_i)$, where $\bar{X}_i$ is the mean of the X_{kj}'s with $k = i$. b is lognormal, so this update requires a Jacobian adjustment to the Metropolis likelihood ratio, and this adjustment is implemented in `acceptanceProbability()`.
- `UniformUpdate`. When `OrderedMCMCParameter`s are updated using only the first three types of updates, there is a tendency for all but the extreme X's in a race to be crammed together. This happens when `ShiftUpdate`s rescale the X's to a narrow scale, then the extreme X's spread away from the bulk due to `IndividualUpdate`s. Now dilations due to `ScaleUpdate`s are discouraged because the extreme X's would be very far from the center. To solve this problem we use `UniformUpdate`s: they leave the extreme X's of a race fixed, but reassign the interior X's using a sample from the uniform distribution (preserving the same order).

Classes which need to be updated in several different ways should be designed in this way with inner classes that implement `MCMCUpdate`. The class then constructs an array of objects that belong to these inner classes, and then the outer class's `update()` method calls the `update()` methods of each of the elements of the array in order to perform all necessary updating. Often the methods of these inner classes that operate on the same class will have some similarity, so that sometimes it will be useful to define a base class which all of these update classes will inherit from.

3 Future Work

YADAS was designed in order to be extensible, there are plans for adding new capabilities. Planned examples include generalized linear models, multivariate normal distribution support, variable subset selection, and model averaging. Some other more fundamental extensions include the following.

3.1 Graphical User Interface

The processes for defining parameters, bonds, and updates are similar enough across applications that YADAS could benefit from a graphical user interface to save users from having to write any Java code at all in many situations. However, it is important that the GUI code generate Java code for a `main()` method, since it is generally easier and more reliable to tweak the analysis using the Java code than it will be to go back through all the manipulations of the GUI. Another possibility is to give the code the capability of loading old analyses so that the analyst can make small changes. The author is also developing a system to do this sort of thing. A desired result is a web-based journal of data analyses in which readers may replay the authors' analyses, and explore whether conclusions change if the analysis is modified, for example by using different covariates, adding another data set, or using the reader's own tool.

3.2 an `MCMC` class

I also intend to make it possible to include an entire statistical model in an object, so that an analysis can (for example) construct an array of these objects and estimate them in turn. An example of a method that this `MCMC` will have is the ability to add new data to the already analyzed data set. There should be other methods which mutate the model in interesting ways. If an intelligent way of adaptively changing the step sizes were available, the software could take over this responsibility if a `changeStepSize()` method existed.

Methods should also be present for changing distributions, removing bonds, changing the group information for a parameter, and undoubtedly many others.

Another reason an `MCMC` class is needed is the "hybrid design" problem in which one uses a genetic algorithm to evaluate expected information gains in a Bayesian model when resources are devoted to several different types of experiments; see Hamada et al (2001). To do this it is important to be able automatically to construct, perform, and evaluate a sequence of analyses.

3.3 an `MCMCMonitor` interface?

I am entertaining the possibility of developing an architecture for monitoring the convergence and other behavior of the MCMC. So far, the code sends output in the form of the values of `MCMCParameters` at each iteration to output files with the same names as the parameters, but the architecture is flexible enough to allow for more general monitoring.

References

[Gelman et al, 1995] Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (1995a), Bayesian Data Analysis, London: Chapman & Hall.

[Gilks et al, 1996] Gilks, W. R., Richardson, S., and Spiegelhalter, D. J. (eds., 1996). Markov Chain Monte Carlo in Practice, London: Chapman and Hall.

[Graves et al, 2001] Graves, T. L., Reese, C. S., and Fitzgerald, M. A. "Analysis of Auto Racing Results." In preparation.

[Hamada et al, 2001] Hamada, M., Martz, H. F, Reese, C. S., and Wilson, A. G. (2001). Finding near-optimal Bayesian experimental designs via genetic algorithms. The American Statistician 55(3): 175-181.

[Hanson and Cunningham, 1999] Hanson, K. M., and Cunningham, G. S. (1999). Operation of the Bayes Inference Engine. In Maximum Entropy and Bayesian Methods, W. von der Linden et al., eds., Dordrecht: Kluwer Academic, pp. 309-318.

[Lunn et al, 2000] Lunn, D. J., Thomas, A., Best, N., and Spiegelhalter, D. (2000). WinBUGS- A Bayesian modelling framework: Concepts, structure, and extensibility. Statistics and Computing 10:325-337.

[Spiegelhalter et al, 1994] Spiegelhalter, D. J., Thomas, A., Best, N. G., and Gilks, W. R. (1994). BUGS: Bayesian inference Using Gibbs Sampling, Version 0.30. Cambridge: Medical Research Council Biostatistics Unit.

[Stern, 1990] Stern, H. S. (1990). Models for Distributions on Permutations. Journal of the American Statistical Association 85: 558-564.